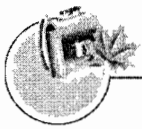

第5章

中断处理

什么是中断？通俗地讲，中断就是你正在干的这件事没干完，突然要停下来去干另一件更加紧急的事。所以，中断就是相对于当前事情的那件更加紧急的事情。

换个角度讲，工作无非分成两种：普通的事和紧急的事。这两种事各有各的特点，所谓普通的事，有的时候也许会很简单，而有的时候却并不简单，一些非常复杂的工作也可以属于普通的事，这些事不是一朝一夕就能完成的，因此，即便耽搁几天也无大碍，而紧急的事，特点就是很急，通常要求你尽快做完。大多数情况下，紧急的事并不复杂，三下五除二就可以完成，但却要求你尽快完成，要不然就会有麻烦。在生活中，这些都是再明白不过的道理了，可要是把这些道理放到操作系统当中去理解，结果却大不一样。

在操作系统当中，我们可以把那些普通的事理解成进程，而把那些紧急的事理解成中断。无论是进程还是中断，对于 CPU 来讲，都是它份内的工作。只不过这两种工作各有各的特点，所以应当区别对待。本章中我们将会涉及操作系统的底层核心之一——中断的处理。

从纯技术的角度出发，每一种 CPU 都有自己独立的中断处理方法。而操作系统又是建立在硬件的基础之上，所以要深究操作系统中的中断处理方法，就不得不提 CPU。另一方面，成熟的操作系统都会尽量摆脱硬件的束缚，为自身或上层应用层提供一种与硬件无关的抽象方法，就像标准库那样，无论在什么硬件平台、什么操作系统下，其编程方法都是统一的，这就是一种很成功的抽象。操作系统中也是处处离不开抽象，中断的处理过程自然也不例外。

在本章，我们需要解决操作系统的中断处理问题，毫无疑问，首先需要介绍 ARM 这种时下最流行的处理器，从它的中断处理方法说起。

5.1　ARM 的中断

对于 ARM 体系结构来说，对中断的处理大致认为可以分成两个阶段：体系统一的处理过程和体系独立的处理过程。所谓体系统一的处理过程指的是同一系列芯片的中断处理方法是一致的，而体系独立的处理过程则是指不同款芯片的处理方法完全不同，我们来举个例子。

以 s3c2410 为例，当中断发生时，芯片首先要保存当前运算环境，比如标志位的值、返回地址，等等。然后，程序跳转到一个固定的地址去执行。上述过程对于整个 ARM9 系列的芯片来说，都是统一的，也就是说无论是 s3c2410 还是 ep9312，或是其他什么芯片，这个过程是完全一致的。CPU 都需要进行返回地址的保存、模式的切换、PC 的跳转等操作。

但是，中断处理就这样完成了吗？显然没有。在 s3c2410 中，如果我们需要处理串口中断，就需要将相应寄存器的 pending 位清除，这样中断才不会重复触发，对于复杂的中断处理过程，我们可能需要局部地禁止某些中断，而同时又允许其他设备产生中断，这些都需要有一个专门的中断控制器来参与工作。而中断控制器属于哪段地址空间、应该怎样控制等问题，都是各芯片的生产商自定义的。例如，在 s3c2410 中，中断控制器位于 0x4A000000 处，而对于 at91rm92 系列芯片，中断控制器则位于 0xFFFFF000 处。所以，同种硬件的中断处理程序在不同的芯片中的处理方法完全不同，也就是说，中断处理程序的这部分工作的处理方法是彼此独立的。

在本章中，中断的这两部分工作我们都会介绍，但显然，体系结构统一的处理方法将会是本章的重点。另外，如不明确指出，本章所涉及到的一些对 ARM 体系结构的描述针对的都是 ARM v7 之前的版本。新版的结构变化较之于传统版本不能说变化不大，不过无论怎样，这对我们能否深入理解嵌入式操作系统的原理不会有太大影响。

5.1.1 统一的异常和中断处理

前面我们大量提到了中断这个概念。但是，如果要深入到技术细节，把ARM的中断处理过程彻底说清楚的话，我们就不得不介绍异常这个概念。

5.1.1.1 ARM的异常

所谓异常，指的就是中止了程序正常执行的过程而不得不去完成的一些特殊工作，如芯片复位、取址失败、指令未定义，等等，具体的异常类型如表5-1所示。

表5-1 ARM的异常状态

异常类型	所属模式	说明
芯片复位	SVC	当芯片复位发生时产生
未定义指令	UND	指令不能被芯片识别时产生
软中断	SVC	软件调用swi指令时产生
预取址中止	ABT	没有权限访问存储器时产生
数据中止	ABT	没有权限访问存储器时产生
中断	IRQ	硬件进行中断请求时发生
快速中断	FIQ	硬件进行快速中断请求时发生

我们通常所说的中断其实也是一种异常，这里的中断包括外部硬件产生的外部中断和由芯片内部硬件产生的内部中断。由中断产生的异常和其他异常，从处理方法的角度来看并没有任何区别，所以我们可以把这些异常统一起来研究。

5.1.1.2 模式与寄存器

首先，我们要从ARM的寄存器开始说起。关于ARM寄存器的一些知识，在前面的章节中已经有过介绍了。在这里，我们会将重点放在各个模式与寄存器的关系中。

我们知道，ARM一共有37个通用寄存器。当然这并不意味着在写程序的时候，可以同时使用这37个寄存器中的任何一个。因为这些寄存器都是根据模式分组的，所以每个模式都有属于各自模式的私有寄存器，程序工作

在某种处理器模式时，只能访问属于该模式的私有寄存器和公有寄存器。这里我们还需要详细说一说另外一个概念，那就是处理器模式。

正如前文中介绍的那样，ARM 一共有 7 种处理器模式，分别是中止模式（ABT）、中断模式（IRQ）、快速中断模式（FIQ）、管理模式（SVC）、系统模式（SYS）、未定义模式（UND）以及用户模式（USR）。其中，除用户模式和系统模式之外的 5 种处理器模式又被称为异常模式，同时，把除用户模式之外的其他 6 种处理器模式称为特权模式。

处理器之所以被设计出支持多种模式是为了能够更好地处理各种异常。

在正常情况下，一个普通程序可能会运行在用户模式或系统模式下，而当中断发生时，ARM 就会自动切换到中断模式去处理中断，处理完成后，又会回到用户模式或系统模式下继续之前的工作。因为每一种模式都包含有相应的私有资源，因此可以保证在处理中断时，原来的程序环境不会被新的环境破坏，从而保证了系统的正常运行。

因为异常有很多种，所以 ARM 处理器在设计时，就定义了 5 种异常模式，分别处理可能出现的异常。当然，将各种异常分得太细，势必会使程序的设计变得复杂，这也是 ARM 异常模式的一个主要缺点。所以在最新的 ARM v7 的体系结构中，人们便打破了这种模式设计的思路，使异常的处理更加简化。当然这并不是我们需要讨论的问题，感兴趣的读者朋友可以进一步查阅相关文档。

对 ARM 寄存器和处理器模式的完整描述如图 5-1 所示。图 5-1 列举了在 7 种处理器模式下，ARM 允许访问的全部寄存器。图中标注有底纹的寄存器代表该模式下的私有寄存器，而未标注底纹的代表共有寄存器。从图中我们可以看出，除用户模式和系统模式之外的处理器模式都有各自的私有寄存器，而用户模式和系统模式自身则共同使用同一组寄存器。

共有寄存器只有一组，或者可以这样解释，所有模式下访问的共有寄存器都是同一个寄存器。也就是说，系统中的 R0、R1、R2、R3、R4、R5、R6、R7、R8、R15 以及 CPSR 各寄存器只有唯一一个。所以，如果在中断模式下和管理模式下都去访问 R0 的话，其实是在访问同一个寄存器。

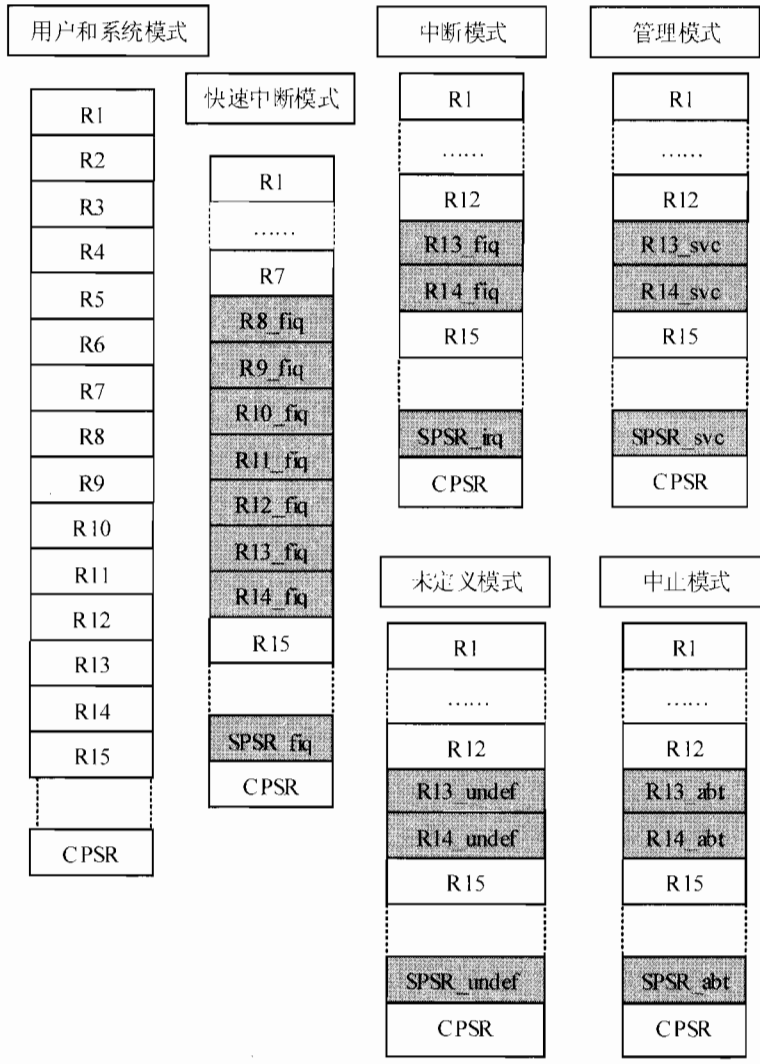

图 5-1　ARM 模式及其寄存器

那么这是不是就意味着，在中断模式或管理模式下，就不能访问 R13 或 R14 了呢？这个问题应该这样理解，单从程序的角度来看，我们可以在中断模式下使用如下代码：

```
mov r14, r2
```

但这里的 R14 指的并不是共有寄存器 R14，而是指中断模式下的私有寄存器 R14_irq。所以上面这条指令的真正意思是：

```
mov r14_irq, r2
```

再比如说，如果有一条指令：

```
mov r0,r13
```

那么在中断模式下，这条指令其实代表了：

```
mov r0,r13_irq
```

而在管理模式里，意思又变成了：

```
mov r0,r13_svc
```

可见没有一种直接的办法能够保证在带有私有寄存器的特权模式下，访问与私有寄存器相对应的共有寄存器的内容。这就为模式切换时对程序的保护提供了一个天然的屏障。

如果从寄存器的应用这个角度来看，除 R15、CPSR 以及各个 SPSR 寄存器以外，所有寄存器并不区分三六九等。但诸如寄存器 R13、R14 寄存器等，还有特殊的作用。这一点在前面的章节中也曾提到过，这里我们再来更详细地介绍一下。

寄存器 R13 用来保存堆栈指针，而寄存器 R14 则保存着上一段程序的返回地址。结合图 5-1，读者朋友们就能理解为什么多数特权模式都含有 R13 和 R14 这两个私有寄存器了。给这些特权模式都赋予一个堆栈指针寄存器，就能保证在这些特权模式下堆栈空间的独立性，避免了互相干扰。而 R14 因为保存有返回值信息，所以如果某种特权模式被动发生了，那么 CPU 会将上一个模式的 R15 寄存器的值保存在相应特权模式下的私有寄存器的 R14 中，从而保证了程序在模式切换时能够正确执行。正是因为寄存器 R13 和 R14 的这两个作用，所以它们都有自己的别名，分别是 SP 和 LR，这样的别称也是可以用在程序里的。

快速中断相对于其他特权模式又稍有不同，除了 R13 和 R14 外，快速中断模式又额外多出了 5 个私有寄存器，这些私有寄存器是专门提供给快速中断模式去迅速地执行中断处理程序的。

通过前面的描述我们知道，共有寄存器是在各个模式下都允许访问的，这在模式切换时会带来一些问题，比如，程序原本运行在系统模式下，寄存器 R0~R12 的值都保存了系统模式下的数据，此时突然发生了中断，程序跳转到中断模式下工作，能够使用的私有寄存器只有 R13 和 R14 以及

SPSR_irq。所以在中断模式下，如果需要使用某些共有寄存器，那就必须先将这些寄存器的值保存到中断模式的堆栈中，这一点我们在前面的章节中也是提到过的。要知道把寄存器压入堆栈是需要花时间的。如果运行在中断模式下的程序需要使用从 R0~R12 的所有寄存器，就需要将这些寄存器全部都压入堆栈。而如果是在快速中断模式下处理中断，又会有怎样的效果呢？如果是在快速中断模式下，最多只需要将寄存器 R0~R7 的内容压入堆栈，从而提高了中断处理的速度。

除了 R13 和 R14 这两个寄存器外，寄存器 R15 也比较特殊。简单地说，它的值代表了当前程序的运行位置。例如，一段程序从内存中 0x0 处开始，至 0x200 处结束，那么当芯片上电时，R15 寄存器的值被初始化为零。CPU 从该寄存器所指向的位置读出第一条指令运行，同时 R15 的值自加 4 个字节。当然，由于多级流水线的设计，R15 的真实情况可能比这要复杂一些，关于这一点我们稍后再做讨论。同样的道理，我们也可以直接改变 R15 的值来实现跳转，如下面这条指令：

```
mov r15,lr
```

该指令可以将保存在 LR 寄存器中的返回值赋给 R15，实现程序的返回。正因为 R15 负担着记录程序代码地址的重任，所以 R15 寄存器也被称为程序计数器，别名是 PC。

另外，CPSR 寄存器以及各模式下的 SPSR 寄存器可以说更加特殊。与 PC 寄存器相同，这些寄存器也不能存储运算结果。CPSR 全名叫做当前程序状态寄存器，其中记录的都是程序当前的运行状态，如当前正在运行的程序属于哪种状态、当前程序的运行结果是否有溢出，等等。不同系列的 ARM 核，CPSR 寄存器的结构可能会有略微的差别，如图 5-2 所示，给出了 ARM 920t 系列中 CPSR 的基本格式。

在图 5-2 中，N、Z、C、V 这 4 个位为条件标志位，某些指令的执行结果会影响这些位，如 cmp。cmp 指令用来比较两个操作数的值是否相等，当二者相等时，CPSR 当中的 Z 位就会变成 1，表示本次运算的结果为 0。有些指令可以在后边加上一个“S”后缀，强行更新标志位，如 subs 指令，就是在做减法的同时，根据结果更新标志位。

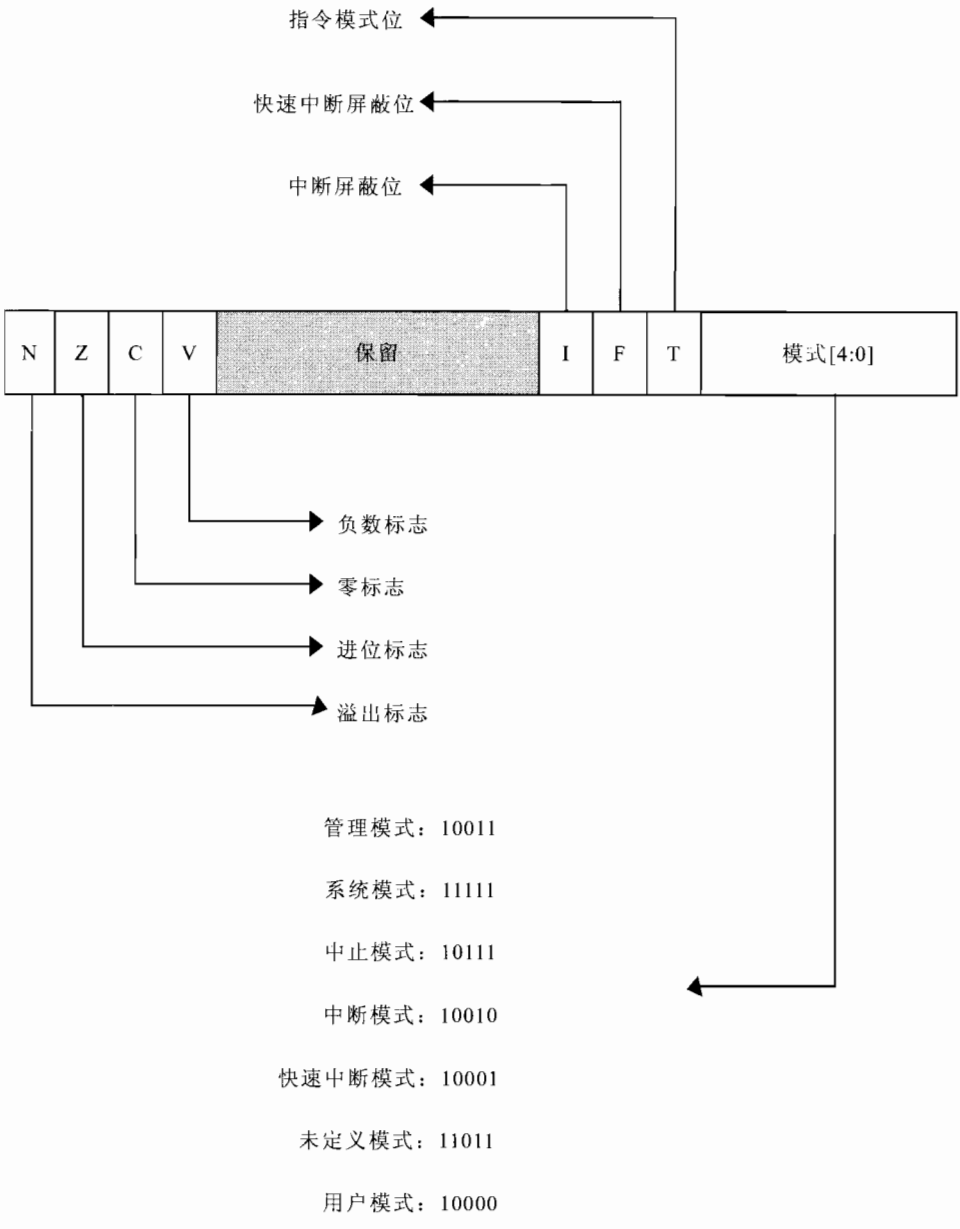

图 5-2　ARM 的 CPSR 寄存器

CPSR 中的 I 位和 F 位分别表示是否允许产生中断和快速中断。相应位的值为 1 代表禁止相应中断，为 0 则代表允许相应中断。在 ARM v6 以前的

体系结构中，并没有提供专门的开关中断指令。因此通常的做法是首先读出CPSR当前值，使用BIC或ORR指令更改相应的位，然后再将新的结果写回CPSR中。

CPSR寄存器的后5位是模式位。当异常模式被触发时，硬件会自动地更新这些位的值，以变换到相应的异常模式。当然，我们也可以手动更改这几个位的值，通过程序实现模式的主动切换。采用的方法同样是“读——修改——写”的经典执行逻辑。

CPSR保证了程序的正确执行，在一个程序正在执行时，无故改变CPSR是非常危险的。但当某个异常发生时，CPU必须中止一段程序的执行，跳转到别处执行另一段程序。如此一来，当异常处理结束，程序跳转回原处时，CPSR的值很有可能就被破坏了。

想要解决这个问题就必须依靠各异常模式的SPSR寄存器。当异常发生时，硬件首先自动地将上一个状态的CPSR寄存器值保存到异常模式下的SPSR寄存器中。而当程序从异常模式返回时，再将SPSR寄存器中的值写进CPSR寄存器。因此，每一种异常模式下都会有一个私有的SPSR寄存器，就像在图5-1中所看到的那样。

以上就是ARM体系结构中有关寄存器、异常以及二者之间的关系的描述。在充分掌握了这些ARM体系结构的基本知识后，我们就可以运用这些知识来学习某一异常发生时CPU会做的工作。

5.1.1.3 异常发生时的处理器动作

当一种异常发生时，硬件就会自动执行如下动作：

（1）将CPSR保存到相应异常模式下的SPSR中。

（2）把PC寄存器保存到相应异常模式下的LR中。

（3）将CPSR设置成相应的异常模式。

（4）设置PC寄存器的值为相应处理程序的入口地址。

以上4个步骤完成后，接下来程序就会进入软件控制部分。剩下的工作就全权交给程序员了。图5-3是对上述4个步骤的总结。

正如前面描述的，当某一异常发生时，硬件自动完成图5-3中所示的4个步骤的工作，软件会从相应异常的入口地址处运行第一条指令。然而，按照我们前面的描述，PC寄存器的值不是应该指向正在运行的指令吗？为什

么图 5-3 中却指向了正在提取的指令，这里的“正在提取”又有什么含义呢？

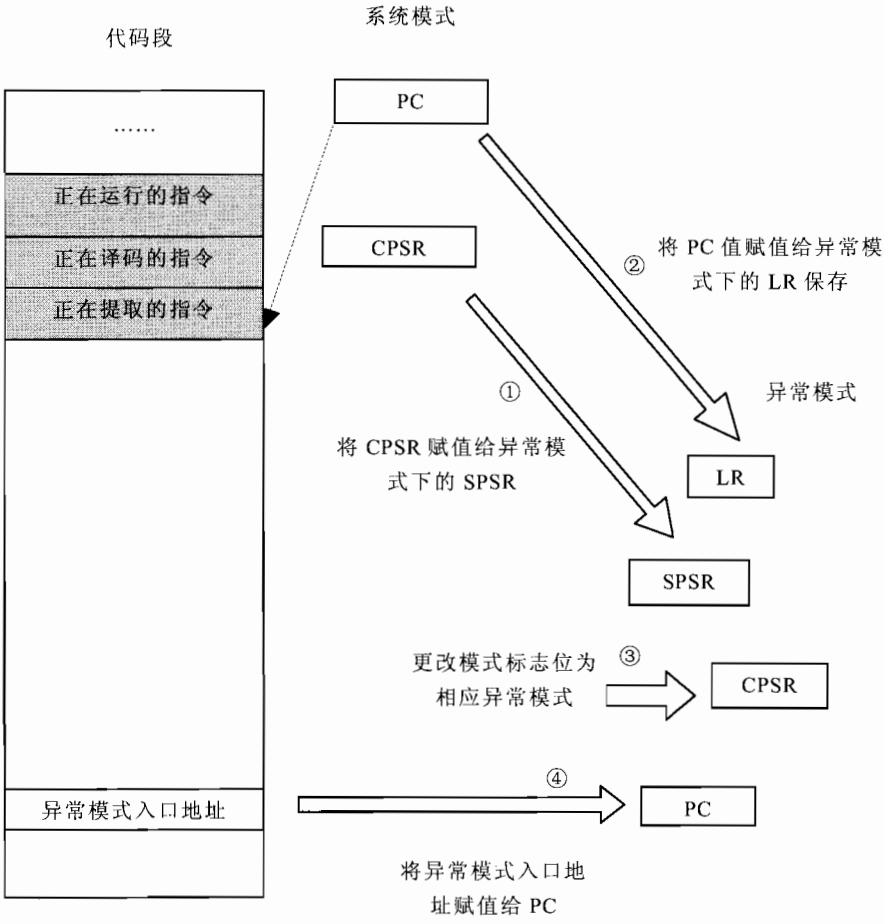

图 5-3　异常发生时，CPU 的动作

这里有一个问题需要说明，因为多级流水线的问题，程序计数器 PC 并不指向当前正在执行的指令，因此读者朋友们可能不能完全理解图 5-3。

要搞清楚这个问题，我们首先需要弄清楚的是一条指令是怎样被执行的。

5.1.1.4　指令执行过程

图 5-4 描述了一条指令的执行过程。

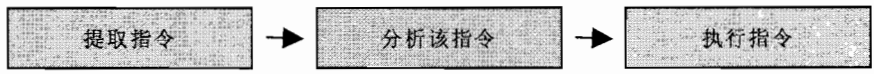

图 5-4　指令的基本执行过程

简单来说，执行某条指令至少要通过取指、译码、执行三个步骤。这就好像盲人在吃饭，第一步是用筷子夹出要吃的东西（从内存中取出指令），第二步是把吃的东西举到鼻子底下闻一下看看是否能吃（分析该指令），第三步是放到嘴里吃（执行指令）。

我们假设该盲人又只有一只手，而每一个步骤又都要一秒钟的时间，那么这位盲人至少要三秒钟才能吃到一样东西，很显然这种吃饭的方法效率太低。所以，如果 CPU 也采取同样的方法，像图 5-4 那样去执行一条指令，那就意味着 CPU 要消耗掉 3 个指令周期才能完成一个动作，可见其运行效率的低下。

为了弥补这个问题，ARM 采用了一种多级流水线的指令执行方式。例如，在 ARM9 中就采用了三级流水线的处理方法，过程如图 5-5 所示。

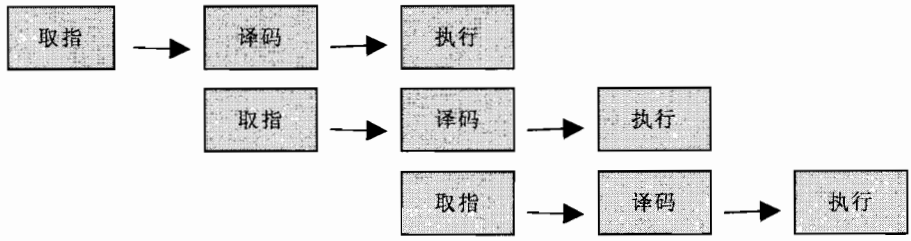

图 5-5　ARM9 中的三级流水线

就像图 5-5 那样，CPU 采用流水线作业的方式，在大多数情况下，是利用三个时钟周期的时间去执行三条指令，从而大大提高了代码运行的效率。

这就好像有一位乐于助人的科学家，知道了盲人吃饭的故事之后，给这位盲人制作了两只机械手，现在盲人已经有三只手了，那么他会怎样吃饭呢？当他的第一只手把吃的送到嘴里吃的时候（执行指令），第二只手已经将另外的食物凑到鼻子底下闻了（分析指令），而第三只手此时正在从盘子里夹第三样东西呢。从此，盲人吃饭的效率就提高了三倍。

现在读者已经理解三级流水的工作原理了，剩下的事情就不言自明了。

程序计数器 PC 的值指向的永远都是待取出的那条指令，也就是说，PC 寄存器的值应是当前正在执行的指令地址加上 8 字节的偏移。到这里，图

5-3 当中的这点疑问也被完全解开了。

此时，如果读者朋友们追问，既然异常模式发生时，最终会转到异常模式入口地址处去运行那里的代码，那么这一入口地址究竟在哪里呢？别着急，这恰恰是我们要向大家介绍的有关异常处理的最后一个知识点——异常向量表。

5.1.1.5　异常向量表

ARM 一共有 5 种异常模式，按道理，每一种异常模式都应该有一个唯一的入口地址。这些入口地址彼此相邻，我们一般称之为异常向量表。

通常情况下，异常向量表是从物理地址 0x0 处开始的，如图 5-6 所示。但这并不是绝对的，我们也可以通过配置 CP15 协处理器，将异常向量表映射到地址的最末尾，即 0xffff000 处。

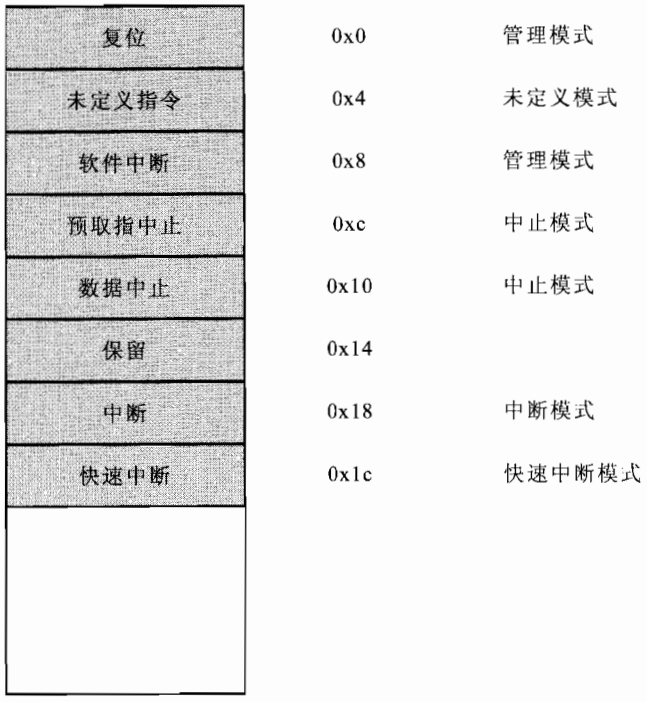

图 5-6　ARM 异常向量表

一个典型的实例是 Linux，在 ARM+Linux 的综合系统中，中断向量表就位于这个位置上。Linux 这么做当然是有道理的，就像我们自己的操作系统那样，Linux 也需要建立虚拟地址到物理地址的映射。但是，Linux 还有另外一条规定，那就是将虚拟地址 0x0~0xbfffffff 的 3G 大小的空间划归为用户空间，而将 0xc0000000~0xffffffff 的 1G 大小的空间定义为内核空间。显然，异常向量表作为系统的一个重要结构，理应归属于内核空间。可是，如果异常向量表按照默认的位置出现在了 0x0 处，就变成了用户空间的资源了。因此，Linux 才需要将中断向量表映射到地址末尾。

异常向量表就是一段出现在固定位置的内存空间，当某一异常发生时，程序最终会到达相应的异常入口去执行存放在那里的指令。但请注意，异常向量表中的每一个入口地址只有 4 个字节大小的空间，因此只能存放一条 ARM 指令。

很显然，这一条指令必须是能够实现跳转功能的指令。实现程序跳转有很多种方法，在启动一章我们也有过介绍。

这样，ARM 与异常向量表有关的代码通常会与下面的代码类似。

代码 5-1

```
_start:
ldr pc,_vector_reset
ldr pc,_vector_undefined
ldr pc,_vector_swi
ldr pc,_vector_prefetch_abort
ldr pc,_vector_data_abort
ldr pc,_vector_reserved
ldr pc,_vector_irq
ldr pc,_vector_fiq

.align 4

_vector_reset: .word   __vector_reset
_vector_undefined:.word   __vector_undefined
_vector_swi:   .word   __vector_swi
_vector_prefetch_abort:   .word   __vector_prefetch_abort
_vector_data_abort:   .word   __vector_data_abort
_vector_reserved: .word   __vector_reserved
_vector_irq:   .word   __vector_irq
_vector_fiq:   .word   __vector_fiq
```

这个时候，我们就要利用之前学到的链接脚本将这段代码链接到内存的 0x0 位置处。这样算来，从_start 开始的 8 条跳转指令刚好可以落在图 5-6 描述的异常向量表正确的位置上。因此，这些跳转指令就是异常向量表的真正内容了。

那么，在代码 5-1 中，在类似于__vector_irq 等这样的变量地址处，存放的正是对相应异常具体的处理方法，就像下面这段代码一样。

代码 5-2

```
__vector_irq:
sub r14,r14,#4
stmfd r13!,{r0-r3,r14}
……
ldmfd r13!,{r0-r3,pc}^
```

stmfd 和 ldmfd 这两条指令，大家应该并不觉得陌生。代码 5-2 首先将 R0、R1、R2、R3 和 R14 这 5 个寄存器的值保存到堆栈中。中断处理的工作完成后，再将各自寄存器的值恢复。有意思的是，当寄存器 R14 从堆栈中恢复的时候，程序并不是直接将堆栈中的数据加载到 R14 中，而是将这个值直接复制给 PC 寄存器。因为 R14 存放的恰是上一段程序的返回值，因此，代码 5-2 的最后一条指令将会在寄存器 R0~R3 的内容被恢复后立刻返回。当然，代码 5-2 作为一段简单的中断处理程序，R14 保存的应该是被中断的那条指令的下一条指令，这也是为什么我们一定要将 R14 的值减去 4 的原因。

中断发生时的 PC 寄存器如图 5-7 所示。

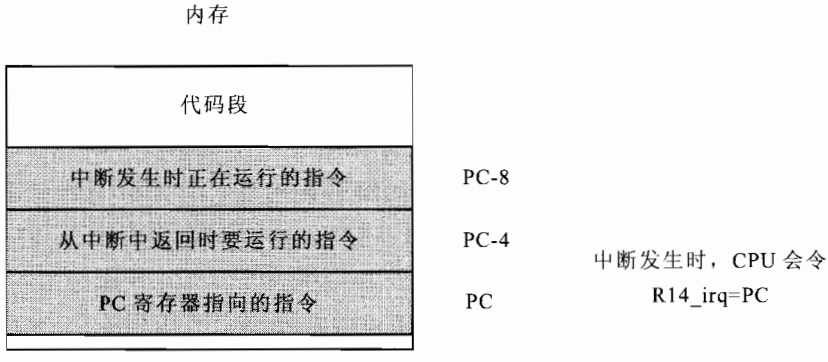

图 5-7　中断发生时的 PC 寄存器

如图 5-7 的描述，中断发生时，CPU 会自动将 PC 寄存器的值保存到中断模式下的私有寄存器 R14 中。中断一旦被执行完，就需要跳转回被中断指令的下一条指令处去执行，这条指令的地址正是 PC-4。因此，我们需要将中断模式下 R14 寄存器的值调整为 R14-4，才是中断结束后真正的程序返回值。

同样的道理也适用于其他异常情况，表 5-2 列出了各种异常模式返回时的调整值。

表 5-2　异常模式

异常	地址	说明
芯片复位	无	未定义芯片复位时的 R14
未定义指令	R14	指向未定义指令的下一条指令
软中断	R14	指向 SWI 指令的下一条指令
预取址中止	R14－4	指向导致预取址中止异常的指令
数据中止	R14－8	指向导致数据中止的那条指令
中断	R14－4	指向被中断的指令的下一条指令
快速中断	R14－4	指向被中断的指令的下一条指令

现在，还剩下一个小细节需要说明。在中断模式下，将 R14 中的值传递给 R15，就跳转回了被中断之前的模式，但此时 CPSR 寄存器中的相应位也需要更改过来，在代码 5-2 中，stmfd 这条指令后边的“^”符号恰恰完成了这个工作。使用这个符号可以在从堆栈中恢复寄存器的同时，将相应模式下的 SPSR 寄存器中的值恢复到 CPSR 中。

此时，ARM 体系结构中的统一的中断处理过程，我们就介绍完了。

5.1.2　独立的中断处理

正像我们前面提到的那样，将 ARM 中断处理过程分为统一的处理过程和独立的处理过程，有助于帮助我们更好地掌握 ARM 下的中断处理方法。虽然在前面的代码中，我们实现了中断的跳转和返回。但此时仍然不能使用中断来解决具体的问题。这一节，我们会以 2410 为例来讲讲与具体硬件相关的中断处理过程。

图 5-8 描述了 s3c2410 的中断产生过程。

要知道，中断产生于外设硬件，也就是说，外设由于某种原因产生了一

个由高电平到低电平或者是由低电平到高电平，又或者是持续的高电平或低电平的信号，而这些信号就会产生一个中断请求。

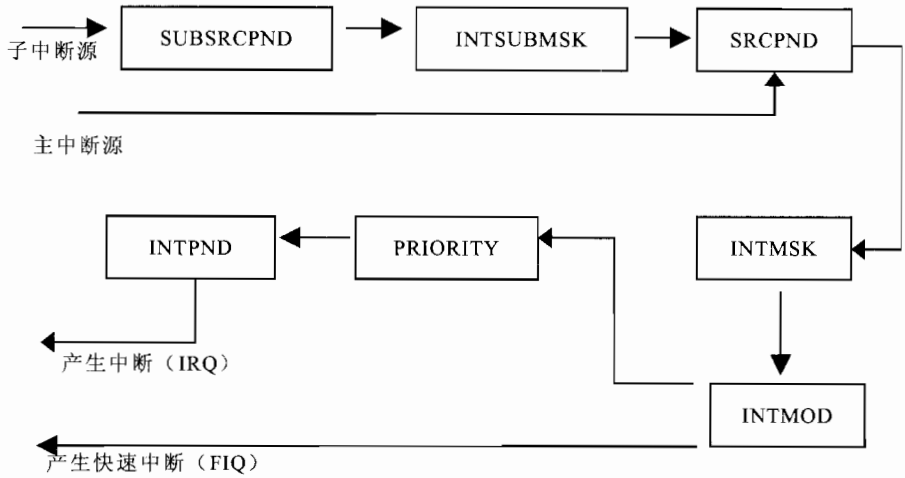

图 5-8　s3c2410 的中断产生过程

当然，请求终归是请求，请求是否真的能被处理还要取决两个因素，一是请求是否被正常传递，二是请求是否被正常处理。

从另一个角度来看，中断本身也可以被分成两种——主中断和子中断。所谓主中断，是指占用一条中断信号线的中断，如 s3c2410 中的时钟中断，该中断独立地占用一条 CPU 中断信号线，即只要 CPU 检测到该中断线有信号，就必然说明时钟产生了一个中断请求，它们之间的关系是一对一的。有的时候，中断和中断之间共同使用同一条信号线，如 s3c2410 中的串口控制器，当串口发送缓冲为零、串口接收缓冲有数据、串口收发错误时，都会在同一条中断信号线上产生请求，即 CPU 只能检测到与串口对应的这条中断信号线上有中断请求，而无法直接就判断出这一中断请求到底是因为哪个原因才产生的。这样，我们就可以将能够产生串口中断的每一种原因叫做子中断了。

说清楚了这两个问题，我们再回头来看一下图 5-8。

在 s3c2410 中，中断都是由中断控制器所控制的，当某一个主中断产生的时候，其结果会保存到中断控制器的 SRCPND 寄存器中。

例如，4 号定时器中断（TIMER4）发生时，SRCPND 寄存器中的第 14

位就会被置 1。从另一个角度讲，CPU 通过判断 SRCPND 寄存器中哪一位被置 1 了就能够知道哪个硬件产生了中断。如果我们有办法人为地将 SRCPND 某一位置 1（当然，这不可能实现），就可以欺骗 CPU，伪装成某个硬件产生中断了。

SRCPND 并不是排他的，也就是说这个寄存器允许同时有多位被置 1，这代表着可以同时有多个硬件产生了中断。

当一个或多个中断同时产生时，中断控制器会将 SRCPND 的内容送去做中断屏蔽检测。对中断屏蔽的控制由 s3c2410 中断控制器中的 INTMSK 寄存器完成。当 INTMSK 寄存器中的某一位为 0 时，这一位代表的硬件所产生的中断就会被传递到下一级，若该位的值是 1，那么该硬件产生的中断就会被屏蔽掉，中断信号就此消失了。

中断屏蔽寄存器 INTMSK 可以筛选掉一些硬件产生的中断，而那些通过了屏蔽寄存器的中断请求将会被送到 INTMOD 寄存器中做模式判断。

INTMOD 寄存器中每一位的默认值都是 0，这表示系统默认会产生 IRQ 中断请求，如果 INTMOD 寄存器中的某一位为 1，则最终会产生 FIQ 中断。FIQ 中断作为一种快速的中断处理，必须要保证它的唯一性。因为如果所有的中断都是快速中断的话，那就相当于没有快速中断了。因此，INTMOD 寄存器在同一时刻有且只能有一位置 1。这就保证了在所有能够产生中断请求的硬件中只允许一个快速中断请求。

当某个中断请求经过 INTMOD 寄存器被确定为快速中断请求时，CPU 将立刻产生一个快速中断，体系结构统一的中断处理过程就会发生。如果经过 INTMOD 寄存器确认为 FIQ 中断请求，则该请求会被送往下一级，做中断优先级的检测，这个过程是由 PRIORITY 寄存器负责的。

很多时候，都会有多个中断请求同时到达 PRIORITY 寄存器。但是，各种各样的硬件中断请求的优先级却各不相同，CPU 会在此阶段选择一个优先级最高的中断请求送往下一级。当然，各种中断请求的优先级是可以根据软件的需求动态调配的，我们可以通过改写 PRIORITY 寄存器的值进行配置。中断请求被送往的下一级是 INTPND 寄存器。

INTPND 寄存器中的每一位代表一个已经通过优先级仲裁的中断请求。因为同一时刻只能有唯一一个中断请求通过 PRIORITY 寄存器，所以，INTPND 寄存器只会有一个位被置 1，我们可以读出这个位来判断究竟是哪

个硬件产生的中断请求。与此同时，CPU 会立刻产生一个中断异常，体系结构统一的中断处理过程就会发生。

至此，从一个硬件产生中断请求到 CPU 触发一个中断异常的全部过程，我们就已经描述清楚了。对于子中断的处理与主中断类似，只是多了两个步骤而已。

在这个过程中，从头到尾都是硬件在做工作。那么从软件的角度来讲，我们又应该做些什么才能保证整个中断处理过程顺利完成呢？

我们知道，当 CPU 接收硬件中断请求并抛出中断异常后，程序会跳转到中断向量表中对应的地址继续运行，最终也会跳转到某段代码处，针对具体的硬件去完成它的工作。在这个过程中，软件至少要完成如下两步才能最基本地保证中断处理的正确性。

（1）我们必须要知道的是，究竟哪个硬件产生了中断请求。一个快速的方法是读取 INTOFFSET 寄存器。该寄存器中的值是位于 0~31 的一个整数，每一个数字代表一个硬件，如 28 代表零号串口。而这个数字与 INTMSK、SRCPND 等寄存器中代表硬件的位序号又是一致的。也就是说，INTPND 寄存器中的第 28 位正是代表了零号串口，该位为 1 则表示这个串口产生了中断。

（2）必须清除掉 SRCPND 和 INTPND 这两个寄存器中代表某一中断请求的相应位。为什么要这样做呢？如果不将相应的中断位置 0，SRCPND 和 INTPND 等寄存器中代表中断请求的对应位将永远是 1，这样，同一个中断请求就会反复产生，而实际上硬件仅仅请求了一次中断，这样一来，错误就产生了。而清除 SRCPND 和 INTPND 两个寄存器的某一位的方法其实很简单，只需要向该位写 1 即可。

在程序中实现以上两点，是保证一个中断被正确处理的关键。表 5-3 列出了 s3c2410 中断控制器的相应物理地址。表 5-4 描述了 s3c2410 中诸如 INTPND 这样的寄存器的各个位所代表的中断源。这些都是在写代码之前需要了解的。

表 5-3　s3c2410 中断控制器

寄存器	地址	默认值
SRCPND	0x4A000000	0x00000000
INTMOD	0x4A000004	0x00000000
INTMSK	0x4A000008	0xFFFFFFFF

续表

寄存器	地址	默认值
PRIORITY	0x4A00000C	0x7F
INTPND	0x4A000010	0x00000000
INTOFFSET	0x4A000014	0x00000000
SUBSRCPND	0x4A000018	0x00000000
INTSUBMSK	0x4A00001C	0x7FF

表 5-4　偏移量代表的中断源

中断源	偏移值	中断源	偏移值
INT_ADC	31	INT_UART2	15
INT_RTC	30	INT_TIMER4	14
INT_SPI1	29	INT_TIMER3	13
INT_UART0	28	INT_TIMER2	12
INT_IIC	27	INT_TIMER1	11
INT_USBH	26	INT_TIMER0	10
INT_USBD	25	INT_WDT	9
Reserved	24	INT_TICK	8
INT_UART1	23	nBATT_FLT	7
INT_SPI0	22	Reserved	6
INT_SDI	21	EINT8_23	5
INT_DMA3	20	EINT4_7	4
INT_DMA2	19	EINT3	3
INT_DMA1	18	EINT2	2
INT_DMA0	17	EINT1	1
INT_LCD	16	EINT0	0

5.2　简单的中断处理实例

接下来让我们亲自运行一段最简单的中断代码，以巩固我们学到的知识。

5.2.1　解决异常向量表的问题

在真正写代码之前，我们还需要完成一件棘手的事情。

回忆一下前面的内容，我们虚拟出来的硬件平台，具有 8M 的 SDRAM，

被挂接到了 s3c2410 的 0x30000000 这个物理地址。之前运行的程序都是以这一假设为基础的。

但如果要运行中断处理程序，异常向量表的问题就不太容易解决了。我们知道异常向量表必须位于内存地址 0x0 处，但是，在我们的虚拟硬件平台中，地址 0x0 根本就不存在任何存储设备。

在实际的应用中，无论是 NAND FLASH 还是 NOR FLASH 都有办法映射到内存地址 0x0 的位置上。因此，将异常向量放在 FLASH 的起始位置，让它们在异常产生时参与程序跳转是一个非常有效的方法。

另一种方法与之类似，我们可以在程序启动时就将已经准备好的向量表搬运到地址 0x0 处。当然，这同样需要有可写的存储器存在。

以上两种方法实现起来并不困难。我们也可以修改虚拟硬件平台的配置来满足这些条件。

```
cpu:  arm920t
mach: s3c2410x

#physical memory
mem_bank: map=M, type=RW, addr=0x00000000, size=0x00100000
mem_bank: map=M, type=RW, addr=0x30000000, size=0x00800000,
file=./leeos.bin,boot=yes
mem_bank: map=I, type=RW, addr=0x48000000, size=0x20000000
```

这样一来，我们就可以将代码的向量表部分搬运到新的存储器中，或者干脆将整个代码段放置于此处，从而实现异常向量表的初始化。

然而，解决异常向量表的方法并不仅限于以上两种。通过 MMU 进行地址重映射，更加灵活方便，本书中就将会使用这种方法来实现中断处理程序。

为了保证异常向量表出现在内存 0x0 的位置上，就必须要修改 MMU 相关的代码，将内存中的中断向量表映射到地址 0x0 处。于是我们需要修改原有代码中的“mmu.c”这个文件，在#define IO_MAP_SIZE 下边一行添加如下内容。

代码 5-3

```
#define VIRTUAL_VECTOR_ADDR          0x0
#define PHYSICAL_VECTOR_ADDR         0x30000000
```

完成相关宏定义后，就需要修改 init_sys_mmu 函数了。请朋友们在该

函数的两个 for 循环之前添加如下一段代码。

代码 5-4

```
for(j=0;j<MEM_MAP_SIZE>>20;j++){
    pte=gen_l1_pte(PHYSICAL_VECTOR_ADDR+(j<<20));
    pte|=PTE_ALL_AP_L1_SECTION_DEFAULT;
    pte|=PTE_L1_SECTION_NO_CACHE_AND_WB;
    pte|=PTE_L1_SECTION_DOMAIN_DEFAULT;
    pte_addr=gen_l1_pte_addr(L1_PTR_BASE_ADDR,\
                             VIRTUAL_VECTOR_ADDR+(j<<20));
    *(volatile unsigned int *)pte_addr=pte;
}
```

以上两段程序负责将内存地址 0x30000000 处开始的 1M 内存映射到地址 0x0 处。这样一来，当中断产生的时候，硬件跳转到中断入口地址 0x18 处执行，而实际运行的则是 0x30000018 处的代码。异常向量表的问题就成功地解决了。

5.2.2 简单的中断处理代码

接下来我们要改写一下启动代码的内容。打开原有代码中的“abnormal.s”文件，原来的代码如下：

代码 5-5

```
__vector_irq:
    nop
```

将其修改为：

代码 5-6

```
__vector_irq:
    sub r14,r14,#4
    stmfd r13!,{r0-r3,r14}
    bl common_irq_handler
    ldmfd r13!,{r0-r3,pc}^
```

在中断模式下，代码最终会跳转到__vector_irq 处运行。此处，CPU 首先会在修正返回地址的同时，保存相关寄存器的数据，然后跳转到函数 common_irq_handler 继续执行。细心的读者可能已经发现了，代码 5-6 其实就是代码 5-2 的具体化。

接下来我们需要新建一个文件，名为“interrupt.c”，添加如下内容。

代码 5-7

```
#define INT_BASE        (0xca000000)
#define INTMSK      (INT_BASE+0x8)
#define INTOFFSET   (INT_BASE+0x14)
#define INTPND      (INT_BASE+0x10)
#define SRCPND      (INT_BASE+0x0)

void enable_irq(void){
    asm volatile (
        "mrs r4,cpsr\n\t"
        "bic r4,r4,#0x80\n\t"
        "msr cpsr,r4\n\t"
        :::"r4"
        );
}

void umask_int(unsigned int offset){
        *(volatile unsigned int *)INTMSK&=~(1<<offset);
}

void common_irq_handler(void){
   unsigned int tmp=(1<<(*(volatile unsigned int *)INTOFFSET));
    printk("%d\t",*(volatile unsigned int *)INTOFFSET);
    *(volatile unsigned int *)SRCPND|=tmp;
    *(volatile unsigned int *)INTPND|=tmp;
    printk("timer interrupt occured\n");
}
```

代码 5-7 包含了三个函数，其中，common_irq_handler 就是代码 5-6 中被调用的那个函数。

在该函数中，程序首先读出了寄存器 INTOFFSET 的值，然后转换成为序号并保存到临时变量 tmp 中。紧接着，又将其写入寄存器 SRCPND 和 INTPND 中，目的是清除寄存器 SRCPND 和 INTPND 中的相应位，这一点我们在上一节的最后部分有详细的描述。最后，函数 common_irq_hander 又打印了一个字符串，表示中断处理被正确地运行。

umask_int 函数负责将相应中断的屏蔽位清除，该函数是通过写 INTMSK 寄存器实现的。

函数 enable_irq 则通过内联汇编的方式清除 cpsr 中的 I 位，全局地使能了中断。

关于代码 5-7，我们还有一点需要强调，因为程序此时已经使能了 MMU，所以代码中使用的寄存器地址就不再是芯片手册中的地址了，而应该是经过映射后的虚拟地址，其基值定义成了宏 INT_BASE。

最后，想要成功编译“interrupt.c”文件，我们还需要将 Makefile 中 OBJS 一行修改为如下形式。

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o interrupt.o
```

到这里，中断处理部分的程序就正式完成了。

下面我们需要一种方法，能够在我们的虚拟设备中产生一些有效的中断，比如每按一次键盘，就会有一个中断在虚拟平台中产生，只有这样，我们才能够看到中断处理程序的运行效果。

当然，键盘中断是一个很好的例子。但最终我们还是选择了定时器中断。究其原因，不仅仅是因为它的配置和使用都相对简单，还因为这个内部硬件通常是作为操作系统时钟周期来驱动进程、定时器等系统核心模块，保证操作系统的正确运行的。

接下来我们就来说一说 s3c2410 中的定时器。

5.2.3 S3C2410 中的定时器

s3c2410 一共集成了五个定时器，TIMER0~TIMER3 四个定时器都有外部引脚的输出，专门为外设提供同步时钟，而第五个定时器 TIMER4 则是一个内部定时器。这五个定时器的工作原理都是一致的，如图 5-9 所示。

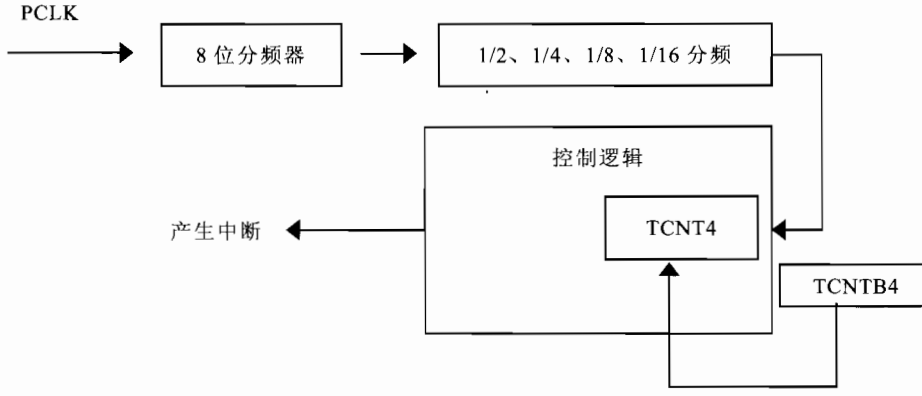

图 5-9 s3c2410 定时器工作原理

首先，定时器所需要的基础时钟由系统的 PCLK 提供。PCLK 是一个由 CPU 核产生的专门供给低速外围设备的基本时钟。通常情况下，程序会选择将 PCLK 的典型值配置成 50M。当然别的值也是可以的，这要根据需求而定。除了 PCLK 以外，s3c2410 还有专门提供给 CPU 内部的时钟 FCLK，提供给高速外设的基本时钟 HCLK 以及专供 USB 设备使用的 UCLK 时钟。FCLK 可以由内部锁相环将外接晶振时钟倍频得到，而 HCLK 和 PCLK 又可以由 FCLK 分频得到。s3c2410 外接晶振的典型频率是 12M，正常工作时，FCLK 的时钟频率往往是 200M 左右，而 FCLK 与 HCLK 和 PCLK 之比一般设置成 1∶2∶4。所以，PCLK 就接近 50M 了，以上过程都可以通过将相应的值写入相关寄存器来实现。

虽然 PCLK 负责驱动慢速外设，但 50M 左右的时钟频率仍然很高。于是，不同的慢速硬件需要根据实际情况对 PCLK 进行再分频，定时器也是如此。s3c2410 的 TIMER 控制器首先将 PCLK 预分频，这是通过将 PCLK 的时钟频率值除以一个 0~255 范围内的某一个整数实现的。

然后，控制器紧接着又对这个结果再次分频，这次分频是将上一步骤的结果除以 2、4、8 或 16 中的某一个数。除得的结果将分别送给五个定时器供它们使用。当然，在这一步中，我们也可以选择使用外接的时钟作为五个定时器的输入时钟，但这种用法并不多见。当二次分频完成时，其结果会送到各定时器的控制逻辑，控制逻辑主要负责计数和比较的工作。

当然，不同定时器的控制逻辑，结构会稍有不同。图 5-9 表示的是 TIMER4 的控制逻辑结构图，也是最简单的一个。在 TIMER4 的控制逻辑内部有一个寄存器 TCNT4。TCNT4 寄存器其实是一个减数器，每当上一个逻辑部件传来一个时钟脉冲时，TCNT4 中的数值就减 1。当减到 0 时，TIMER4 就会产生一个中断请求。然而，TCNT4 这个寄存器是不可见的，也就是说，我们不能直接读写这个寄存器的内容。想要改写 TCNT4 寄存器的初始值，必须通过改写外部寄存器 TCNTB4 来实现，当我们将某一个数写入到 TCNTB4 寄存器时，该值并不会立刻被复制到内部寄存器 TCNT4 中，而是要等到 TCNT4 中的值减到零时才会复制。

以上就是 TIMER4 定时器的硬件执行流程。从软件的角度来讲，我们也只需要依据上述过程对相关的寄存器进行配置，就可以完成对定时器的操作。表 5-5 列出了与 TIMER4 有关的寄存器及其使用方法。

表 5-5 s3c2410 中与定时器有关的寄存器

寄存器	地址	说明
TCFG0	0x51000000	15:8 位代表 TIMER4 一级分频器的值
TCFG1	0x51000004	23:20 位可以选择使用 DMA 方式还是中断方式，19:16 位代表二级分频器的值
TCON	0x51000008	22:20 位用来配置 TIMER4
TCNTB4	0x5100003C	代表 TIMER4 减数寄存器的数值

我们将具体的步骤总结如下。

（1）根据 PCLK 的值确定一级分频器和二级分频器的具体值，改写 TCFG0 和 TCFG1 两个寄存器。TCFG0 只需要向 15:8 写一个 8 位数即可，TCFG1 的默认值是中断方式工作、1/2 分频，这个可以改写，也可以使用默认值。

（2）根据需要配置 TCNTB4，调整减数器的值。

（3）配置 TCON 寄存器。以 TIMER4 为例，TCON 寄存器的第 22 位代表用来配置是否使用 TIMER4 的 autoreload 功能。所谓 autoreload，是指控制逻辑中的内部寄存器 TCNT4 在减数到零时，是否自动地从 TCNTB4 寄存器中将新的值自动地复制到 TCNT4 里。想要定时器持续有效，必须开启 autoreload 功能，也就是将 TCON 的第 22 位置 1。将 TCON 的第 21 位置 1，会使 TCNTB4 中的内容加载到 TCNT4 中，在首次使用 TIMER4 时，这一步是需要的。在 autoreload 模式下，如果已经将 TCNTB4 中的内容复制到 TCNT4 了，就可以把 TCON 的第 21 位清零了。TCON 的第 20 位用来控制 TIMER4 的开始和停止，可以通过将该位置 1 来启动 TIMER4。

现在，所有关于定时器的原理和操作方法我们就介绍完了。

5.2.4 让中断处理程序运行起来

事不宜迟，让我们来实现一下这段代码吧！

代码 5-8

```
#define TIMER_BASE  (0xd1000000)
#define TCFG0   ((volatile unsigned int *)(TIMER_BASE+0x0))
#define TCFG1   ((volatile unsigned int *)(TIMER_BASE+0x4))
#define TCON    ((volatile unsigned int *)(TIMER_BASE+0x8))
#define TCONB4  ((volatile unsigned int *)(TIMER_BASE+0x3c))
```

```
void timer_init(void){
    *TCFG0|=0x800;
    *TCON&=(~(7<<20));
    *TCON|=(1<<22);
    *TCON|=(1<<21);
    *TCONB4=10000;
    *TCON|=(1<<20);
    *TCON&=~(1<<21);

    umask_int(14);
    enable_irq();
}
```

函数 timer_init 首先对与 TIMER4 相关的寄存器进行了适当的配置。然后，利用 umask_int 函数清除掉中断屏蔽寄存器中代表 TIMER4 的那一位。最后，使用 enable_irq 函数使能全局中断。整个过程完全是按照前面描述的方法依次进行的。

同样的，这里与 TIMER4 寄存器有关的地址使用的仍然是虚拟地址。因此，寄存器地址偏移值应该是 0xd1000000，而不是表 5-5 中描述的 0x51000000。

我们可以将代码 5-8 的内容保存到“boot.c”中的 plat_boot 函数前面，然后修改 plat_boot 函数为如下形式：

代码 5-9

```
void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    test_printk();
    timer_init();
    while(1);
}
```

所有代码都修改完毕了，此时可以在终端运行 make 命令，如果成功编译，运行 skyeye 命令，你将看到如下输出：

```
helloworld
test_mmu
testing printk
test string :::    this is %s test
test char ::: H
test digit ::: -256
test X ::: 0xffffff00
test unsigned ::: 4294967040
test zero ::: 0
14  timer interrupt occured
14  timer interrupt occured
14  timer interrupt occured
14  timer interrupt occured
14  timer interrupt occured
14  timer interrupt occured
.....
```

终端会不停地输出字符串“timer interrupt occurred”，表示 TIMER4 中断被定时触发。至此，在我们自己的操作系统中也能实现基本的中断了。

5.3 复杂的中断处理实例

在前面的描述中，为了突出重点，我们忽略了很多细节。在本节中，我们将进一步优化这段程序代码，这需要先从分析被我们忽略的那些细节开始了解这些细节所带来的问题，然后再想办法解决它们。

5.3.1 提高中断的效率

首先有一点必须要强调，能够产生中断请求的硬件有很多，这些硬件什么时候产生中断请求，我们并不清楚。也就是说，当我们正在处理某个硬件产生的中断请求时，另外的硬件可能也产生了中断请求。那么当这种情况发生时，使用我们前面介绍的简单的中断处理方法会带来怎样的问题呢？我们先来分析一下。

中断的简单执行过程如图 5-10 所示。

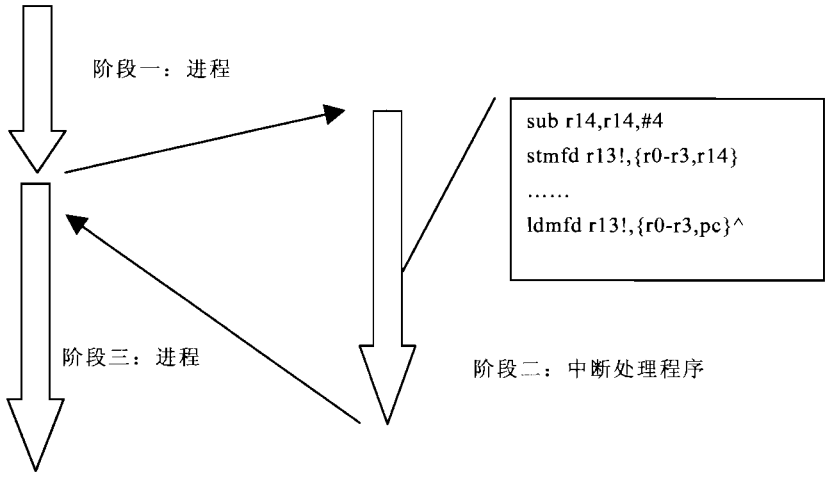

图 5-10　中断的简单执行过程

如图 5-10 所示，可以将中断的简单执行过程划分为三个阶段。某个进程在第一阶段运行时，被一个中断中止，于是系统进入第二阶段，进行中断的处理，处理完成后，再回到进程被中止的位置继续执行，运行第三阶段的代码。

正常情况下，程序运行在第一阶段和第三阶段时，是允许被其他的中断中止的。但当第二阶段发生时，CPU 会自动地将 CPSR 寄存器的 I 位和 F 位置 1。也就是说，在第二阶段运行时，中断默认是被禁止的。此时若有其他硬件产生了新的中断请求，则该中断请求将不会被立刻处理，而只能等到退出第二阶段后再去处理。

这样，问题就来了。如果第二阶段程序运行时间较长，并且同时有很多的硬件产生了中断请求的话，就会出现中断的延迟，导致某些中断等待了过长的时间才被处理，违背了中断处理程序的初衷。在实际的运行环境下，这种情况还是经常会发生的。

上面我们对中断处理过程中可能遇到的普遍问题进行了描述，这个问题即是中断延迟。所谓中断延迟，指的是从硬件产生中断请求开始到运行中断处理程序中的第一条指令之间的间隔时间。中断处理的基本原则是能够在产生请求时，第一时间对中断进行处理。而中断延迟恰恰延长了这段时间，其结果是使系统响应速度变慢，严重时甚至会引起系统瘫痪。因此，一个以应

用为目的的操作系统必须要想办法最小化中断延迟，来提高系统的运行效率和稳定性。

解决这个问题的方法通常有两个，一是允许中断嵌套，二是引入中断优先级，优先处理高优先级的中断。

使用中断嵌套来处理中断延迟的方法多见于一些通用操作系统之中。中断嵌套的基本思想是，中断请求不分高地贵贱，生来平等，没有谁重要谁不重要之分，只优先处理最新发生的中断请求。因此，通过中断嵌套的方法处理中断延迟问题能够在一定程度上解决中断延迟问题。

图 5-11 是中断嵌套的示意图。

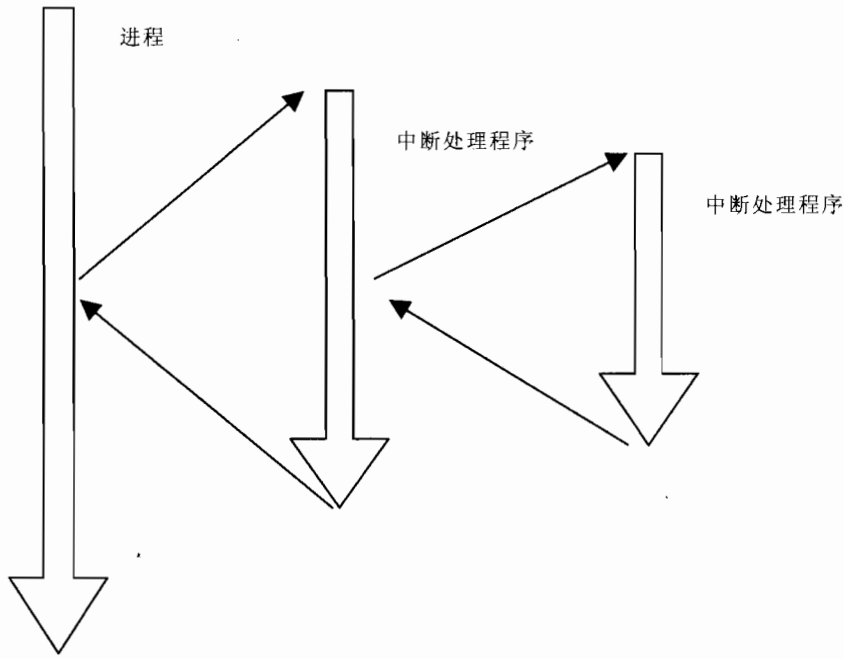

图 5-11　中断嵌套示意图

如图 5-11 所示，当第一个中断处理程序还没有执行完毕的时候，第二个中断请求就已经产生了，于是 CPU 去执行第二个中断处理程序，执行完毕后，又回到第一个中断处理程序中继续执行，这就是一个最简单的中断嵌套模型。

使用这种方法，第二个中断请求的延迟就完全没有了，而第一个中断请求虽然执行的时间被拖长了，但毕竟也执行了一部分，从而使整体的中断运行效率提高。

图 5-11 看似简单，但是在真正执行的时候，却没有那么容易。首先我们要解决的是，怎样在一个中断正在被处理的过程中允许另外一个中断发生。有的读者朋友可能会觉得这很容易，用 msr 和 mrs 清除掉 CPSR 寄存器的 I 标志就可以了。事实当然不会那么简单，读者朋友们可以结合图 5-3 看一下在中断模式下，正在执行中断处理程序时又发生了中断，会有什么严重后果。

（1）在中断模式下，寄存器 R14_irq 作为私有寄存器，保存着被中止的进程的返回地址。我们将它保存到堆栈当中去，目的是当中断模式发生函数调用时，该寄存器可以用来保存函数返回值。但如果此时另一个中断恰好发生，R14_irq 就又要保存新的中断处理程序返回时的地址，原来的返回值便会丢失。

（2）同样地，在中断模式下，SPSR_irq 寄存器保存了上一个模式的 CPSR 值，但是如果在中断模式下再次发生中断，那么 SPSR_irq 又要保存新中断的 CPSR，从而将原来的 CPSR 丢掉。

图 5-12 描述的就是以上两个问题。我们可以看到，如果中断模式下又发生了中断，相应的寄存器所保存的值就会被更新，从而造成数据丢失。即便我们可以将 SPSR_irq 寄存器的值保存到堆栈中，避免了该寄存器被篡改，也无法通过类似的手段解决寄存器 R14_irq 的问题。

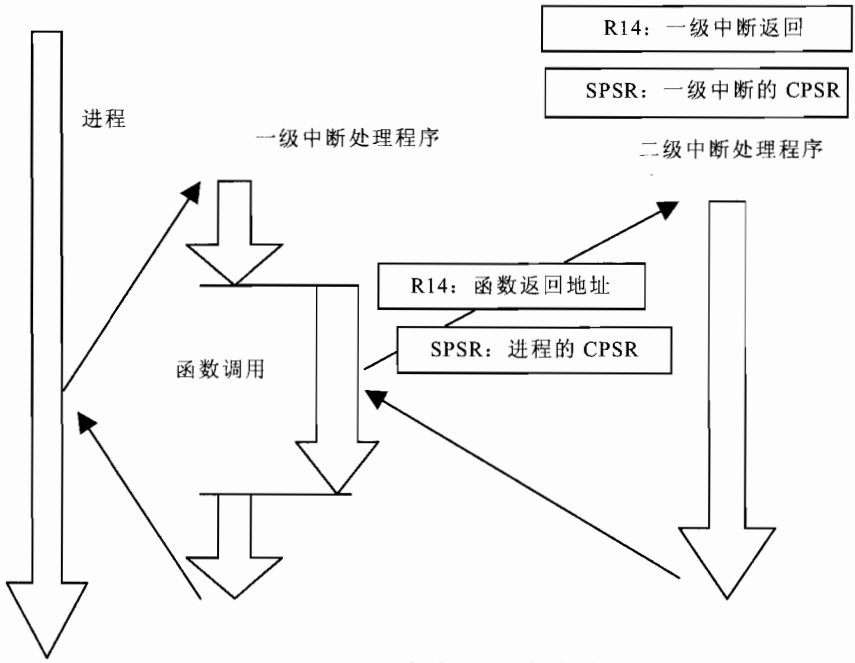

图 5-12　中断嵌套时，寄存器的问题

当然，难解决并不代表不能解决，只不过我们需要换个思路，不在中断模式下，而是切换到其他模式进行中断处理。

当中断处理程序发生在其他模式（如 SVC 模式）时，函数调用的返回值会保存在寄存器 R14_svc 中，即使此时中断再次发生，CPU 也会自动进入中断模式，将被中断的程序的返回值保存在 R14_irq 中，二者不再冲突。这就是实现中断嵌套的普遍方法。

5.3.2 中断嵌套的实现

让我们再一次通过例子来深入理解一下中断嵌套。我们需要将文件“abnormal.s”中的内容修改一下，如下所示：

代码 5-10

```
.equ DISABLE_IRQ,0x80
.equ DISABLE_FIQ,0x40
.equ SYS_MOD,0x1f
.equ IRQ_MOD,0x12
.equ FIQ_MOD,0x11
.equ SVC_MOD,0x13
.equ ABT_MOD,0x17
.equ UND_MOD,0x1b
.equ MOD_MASK,0x1f

.macro CHANGE_TO_SVC
      msr  cpsr_c,#(DISABLE_FIQ|DISABLE_IRQ|SVC_MOD)
.endm

.macro CHANGE_TO_IRQ
      msr  cpsr_c,#(DISABLE_FIQ|DISABLE_IRQ|IRQ_MOD)
.endm

.global __vector_undefined
.global __vector_swi
.global __vector_prefetch_abort
.global __vector_data_abort
.global __vector_reserved
.global __vector_irq
.global __vector_fiq
```

```
.text
.code 32

__vector_undefined:
        nop
__vector_swi:
        nop
__vector_prefetch_abort:
        nop
__vector_data_abort:
    nop
__vector_reserved:
    nop
__vector_irq:
    sub r14,r14,#4
    stmfd r13!,{r14}
    mrs r14,spsr
    stmfd r13!,{r14}
    CHANGE_TO_SVC
    stmfd r13!,{r0-r3}
    bl common_irq_handler
    ldmfd r13!,{r0-r3}
    CHANGE_TO_IRQ
    ldmfd r13!,{r14}
    msr spsr,r14
    ldmfd r13!,{pc}^

__vector_fiq:
    nop
```

代码 5-10 是在原有内容的基础上修改了函数__vector_irq 的部分而得到的。将原来的简单中断处理方式修改成现在的嵌套中断处理方式，其原理正是我们前面所介绍的更换处理模式的方法。

首先，我们需要正确地计算程序上一个状态的返回值并及时地将该返回值压入堆栈。这是通过 sub r14,r14,#4 和 stmfd r13!,{r14}这两条指令实现的。

然后，mrs R14,spsr 这条指令负责将 SPSR_irq 寄存器的值保存到 R14 中，因为 R14 寄存器的值已经被压入堆栈了，因此这里利用它作为一个普通寄存器来使用是安全的。此时，R14_irq 保存的就是中断状态下的 SPSR 寄存器的值，同时也是上一个被中断的状态的 CPSR 寄存器的值，此时也将

该值压入中断模式堆栈中。这样，在保护好上一个状态的 CPSR 和返回值后，即使中断再次发生，这两个值也不会遭到破坏，从而保证了程序执行的正确性。

在相关值保存完毕后，我们就可以跳转到 SVC 模式了。这么做的目的是为了保证在函数调用时，返回值可以保存在 R14_svc 中，而不是原先的 R14_irq 中。这样，一旦中断恰在此时发生，R14_irq 就可以用来保存上一状态的返回值，避免函数空间被破坏。

此处，我们是利用宏定义的方式实现模式间的切换的。关于汇编模式下的宏定义方法，下面介绍一下。

GNU 汇编下的宏定义的基本格式如下：

代码 5-11

```
.macro THE_NAME_OF_YOUR_MACRO (PARAMETER)
        assamble code
.endm
```

如代码 5-11 所示，宏是以.macro 关键字开始的。THE_NAME_ OF_YOUR_ MACRO 代表了宏定义名字，如果宏有参数，参数要跟在宏名的后边。紧接着若干行是实际的汇编代码。最后一行以.endm 关键字结尾，代表宏定义到此结束。

代码 5-12

```
.macro CHANGE_TO_SVC
        msr  cpsr_c,#(DISABLE_FIQ|DISABLE_IRQ|SVC_MOD)
.endm
```

代码 5-12 就是一个宏定义的最简单例子。在该例子中，CHANGE_TO_SVC 就是宏定义的名字，msr 指令一行就是该宏定义的具体内容。这个宏非常简单，只有一行汇编代码。

如果想在宏定义中使用参数，可以参考代码 5-13。

代码 5-13

```
.macro  enable_fiq      reg
        mrs \reg,       cpsr
        bic \reg,       \reg,   #DISABLE_FIQ
        msr cpsr,       \reg
.endm
```

在代码 5-13 中，reg 是宏 enable_fiq 的参数，而在汇编代码中，对该参

数的引用是通过将该参数名字前面加上一个“\”符号来实现的。这样，在使用这个宏定义时，就可以通过某个寄存器名来替换 reg 参数，例如：

```
enable_fiq r0
```

这样一来，关于 GNU 汇编下宏定义的使用方法我们就介绍完了。此时再回到代码 5-10 中，__vector_irq 函数正是通过宏 CHANGE_TO_SVC 实现了从中断模式向管理模式的切换，同时禁止中断和快速中断发生。

之后，寄存器 R0 ~ R3 的值被保存到管理模式下的堆栈之中。在做好了所有准备工作之后，程序调用 common_irq_handler 函数进行具体的中断处理。

函数 common_irq_handler 与代码 5-7 中的函数也稍有不同，其定义如下：

代码 5-14

```
void common_irq_handler(void){
   unsigned int tmp=(1<<(*(volatile unsigned int *)INTOFFSET));
    printk("%d\t",*(volatile unsigned int *)INTOFFSET);
    *(volatile unsigned int *)SRCPND|=tmp;
    *(volatile unsigned int *)INTPND|=tmp;
    enable_irq();
    printk("interrupt occured\n");
    disable_irq();
}
```

在代码 5-14 中，由 printk("interrupt occured\n")一行所代表的具体硬件的中断处理代码前后各添加了一条新的代码，分别是使能和禁止全局中断。也就是说，在函数 enable_irq()和 disable_irq()之间，虽然与某一硬件有关的中断程序正在处理，但同时也允许其他嵌套中断发生。

在调用完 common_irq_handler 函数之后，需要将 R0 ~ R3 四个寄存器的值从堆栈中弹出。之后程序调用 CHANGE_TO_IRQ 宏，再次返回到中断模式。在中断模式下弹出堆栈中保存的 SPSR 值之后，返回到中断前的状态中。

这样，一个可嵌套的中断处理过程就正式完成了。

运行这段代码的方法其实也很简单。读者用代码 5-10 和 5-14 的内容替换掉原有的代码，之后编译并运行，结果如下：

```
helloworld
test_mmu
testing printk
```

```
test string :::    this is %s test
test char ::: H
test digit ::: -256
test X ::: 0xffffff00
test unsigned ::: 4294967040
test zero ::: 0
14  interrupt occured
14  interrupt occured
14  interrupt occured
14  interrupt occured
14  interrupt occured
......
```

5.4 更优秀的中断嵌套方法

代码5-10中使用的中断嵌套方法虽然能够很好地解决中断延时的问题，但仍然有不足之处。其中一点就是中断被使能的时间偏少。

我们看到，在代码 5-10 中，使能中断发生在 common_irq_handler 函数中，清除相应中断标志之后，被使能了的中断很快就又被禁止了，在调用 common_irq_handler 函数之前和调用之后，中断都是被禁止的。

在中断处理过程中，使能中断时间的长短决定了系统实时性的优劣。因此，优秀的操作系统都要求可嵌套中断尽量延长使能中断的时间。本节中我们介绍一种被广泛采用的方法，可以尽可能地延长中断使能时间，保证系统的强实时性。

仔细分析一下代码 5-10，在处理中断请求时，程序首先从中断模式切换到管理模式，在管理模式下使能全局中断，在中断处理完成后，又需要先禁止全局中断，然后再切换回中断模式，最后返回到上一状态中。

结合以上分析以及前面我们对可嵌套中断所产生的问题的描述，读者会发现，中断嵌套的一个原则是不允许在中断模式下使能全局中断。因为这样会导致中断模式下的函数返回值寄存器 R14_irq 有可能被改写。所以，想要延长中断处理过程中使能中断的时间，就意味着要尽量减少中断处理程序在中断模式下的工作时间。

如果我们能够让中断处理程序在管理模式下直接返回到被中断之前的模式，而不是像代码 5-10 那样从中断模式返回，就可以在一定程度上延长中断使能的时间，从而提高系统实时性。

代码 5-10 在中断时的模式切换过程如图 5-13 所示。

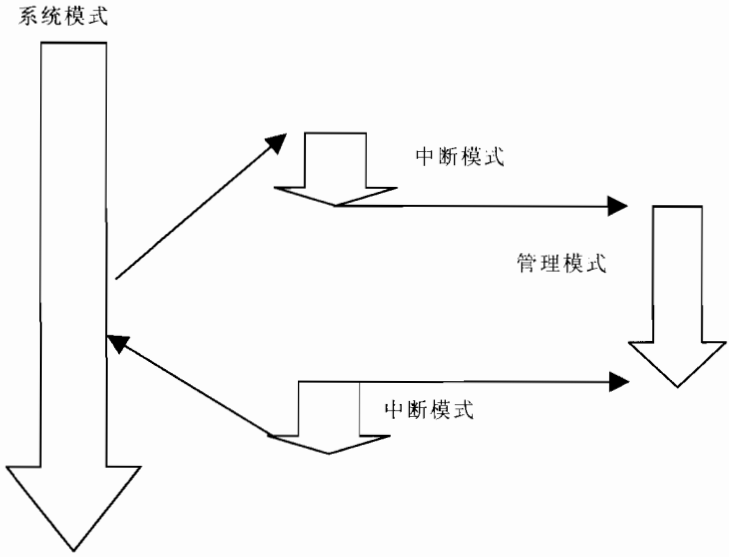

图 5-13　简单的中断嵌套方法

与图 5-13 相比，图 5-14 中的中断模式出现的时间更短。因此，在中断处理过程中，中断会有更多的时间处于使能状态，这样就延长了中断嵌套的时间，从而提高了系统运行效率。

实现该功能的关键一点是将上一个状态的相关寄存器直接保存到管理模式的堆栈中，而不是像代码 5-10 那样在中断模式下保存。处理好这个问题后，其他的就很简单了。

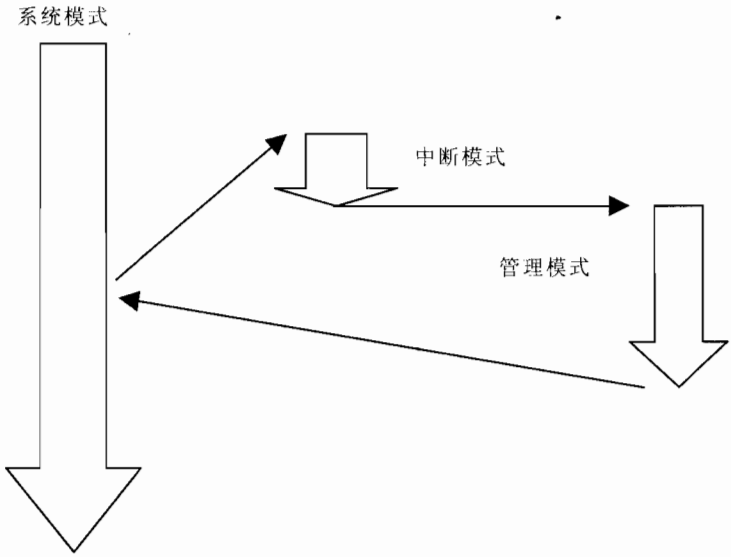

图 5-14　快速实现中断嵌套的方法

接下来，就让我们看一下按照图 5-14 的流程改写的中断处理程序代码。

代码 5-15

```
__vector_irq:
        sub r14,r14,#4
        str r14,[r13,#-0x4]
        mrs r14,spsr
        str r14,[r13,#-0x8]
        str r0,[r13,#-0xc]
        mov r0,r13
        CHANGE_TO_SVC
        str r14,[r13,#-0x8]!
        ldr r14,[r0,#-0x4]
        str r14,[r13,#-0x4]
        ldr r14,[r0,#-0x8]
        ldr r0,[r0,#-0xc]
        stmdb r13!,{r0-r3,r14}
        bl common_irq_handler
        ldmia r13!,{r0-r3,r14}
        msr spsr,r14
        ldmfd r13!,{r14,pc}^
.....
```

在代码 5-15 中，第一步的工作仍然是计算返回地址，对于中断程序，返回地址的处理方法都是一致的。

紧接着，程序再将该返回地址保存到一段内存当中。返回地址被保存到哪块内存并不重要，重要的是该内存地址不能被别的进程破坏，并且同时允许管理模式访问，这样程序才能在管理模式中把这个返回地址读出来。

我们当然可以像确定异常堆栈那样，预先在系统初始化时就划分出块独立的空间用来保存。不过，更好的办法是像代码 5-15 那样，将该值暂时地保存到中断模式堆栈的低地址处。因为堆栈指针都是向下生长的，类似于 R13_irq-#0x4 这样的地址，都是没有被使用过的地址，因此可以用来保存临时数据。

同样的方法也适用于 SPSR 寄存器，也就是上一个状态的 CPSR 寄存器的值。代码 5-15 首先将这个值赋给 R14_irq，然后保存到 R13_irq-#0x8 处，因为 R14_irq 的值已经备份过了，因此这里可以将它当成是一个普通寄存器使用。

然后，我们还需要将 R0 的值保存到 R13_irq-#0xC 处，这是因为 R0 这个寄存器需要保存 R13_irq 的值。通过 R0 寄存器可以将中断模式下的堆栈指针传递给管理模式。

当前面这些工作完成后，调用 CHANGE_TO_SVC 切换到管理模式。在管理模式下，代码 5-15 首要的任务是恢复寄存器，要恢复的寄存器包括：

- 被中断的状态的返回值
- 被中断的状态的 CPSR 值
- R0 寄存器

它们分别被保存在 R13_irq-#0x4、R13_irq-#0x8 和 R13_irq-#0xC 三个地址中。因为在中断模式下，R13_irq 已经被保存到 R0 中，这样，在管理模式里，我们可以通过 R0-#0x4、R0-#0x8 和 R0-#0xC 来读取它们。读出这些值后，还需要按照一定的要求对它们正确地进行压栈。压栈的顺序自顶向下分别是被中断状态的返回值、管理模式原来的 R14 值、被中断状态的 CPSR 值，还有 R0~R3 寄存器的值。

当压栈完成后，程序立刻调用函数 common_irq_handler 进行具体的中断处理工作。之后，程序依次恢复各个寄存器，并从管理模式直接返回到中

断之前的运行状态。

代码 5-15 实现了图 5-14 的执行流程。这样当程序切换到管理模式后，就可以直接开启全局中断，直至最后返回，全局中断都不需要被禁止，从而能够更大限度地提高中断嵌套的效率。

当然，为了配合代码 5-15，对 common_irq_handler 函数要稍做修改。

代码 5-16

```
void common_irq_handler(void){
   unsigned int tmp=(1<<(*(volatile unsigned int *)INTOFFSET));
    printk("%d\t",*(volatile unsigned int *)INTOFFSET);
    *(volatile unsigned int *)SRCPND|=tmp;
    *(volatile unsigned int *)INTPND|=tmp;
    enable_irq();
    printk("interrupt occured\n");
}
```

在代码 5-16 中，从 enable_irq()函数开始直至整个中断处理过程的结束，全局中断都是被允许的，程序中不需要调用 disable_irq()函数。

使用同样的方法运行代码 5-15 和代码 5-16，我们将会看到与前面一致的结果。

```
helloworld
test_mmu
testing printk
test string :::   this is %s test
test char ::: H
test digit ::: -256
test X ::: 0xffffff00
test unsigned ::: 4294967040
test zero ::: 0
14  interrupt occured
14  interrupt occured
14  interrupt occured
14  interrupt occured
14  interrupt occured
.....
```

一个更加优秀的中断处理方法就大功告成了！

5.5　总结

至今还记得一篇小学课本上的文章，是由世界著名数学大师华罗庚写的《统筹方法》，要解决的问题没有变，但是经过对操作步骤和结构的适当调整后，工作效率就被大大提高了。正如文章中所说的那样，统筹方法是一种安排工作进程的数学方法，它的实用范围极其广泛，在操作系统的设计中也不例外。

正是因为这种基本思想的出现，在芯片级，人们便设计了两种执行逻辑，就是本章一开始提到的进程和中断。这两种执行逻辑共同努力、各司其职，从而整体地提高了系统的运行效率。

同样，在进程或中断内部也无时无刻不充斥着科学的设计思想。结合本章的例子，我们也可以清楚地看到，同为中断处理程序，依托于某种硬件平台，采用不同的处理方法，最终能够达到的效果也不尽相同。

计算机科学不是局限在程序、语言上的一种小技巧，而是从抽象的算法分析、形式化语法到软硬件等的一个系统的科学门类。